Pergamon

*Computers Math. Applic.* Vol. 29, No. 1, pp. 115–121, 1995

0898-1221/95 $7.00 + 0.00

0898-1221(94)00211-8

# An Artificial Intelligence Approach for the Generation and Enumeration of Perfect Matchings on Graphs

M. M. BALAKRISHNARAJAN AND P. VENUVANALINGAM[1]
Department of Chemistry, Bharathidasan University
Tiruchirappali, 620 024, India

*(Received and accepted August 1993)*

**Abstract**—A new search based algorithm is developed for an effective combinatorial exploration of all the perfect matchings on a general graph. The algorithm makes use of a relations based representation of the graph and is completely independent of the nature of the graph. Though this is a typical $NP$-class subgraph enumeration problem, the search ranks high in selectivity without compromising accuracy using heuristic guidance. Since the present algorithm is quite compact, nonspecific and generates all perfect matchings of the graph, it excels all the earlier algorithms in its uniqueness. Further it is flexible to accommodate domain specific efficiency improvements, which is illustrated for some classes of graphs.

## 1. INTRODUCTION

Let $G(V, E)$ be a finite graph. A matching $m$ on $G(V, E)$ is any subset of edges $\{e \in E \mid \forall v \in E \deg(v) = 1\}$. A matching $m$ with $k$ edges that are not adjacent is called a $k$-matching. Here, $k$ is referred as the cardinality of $m$ denoted by $|m|$. A maximal matching $Mm$ of $G(V, E)$ is a matching to which no edge $e \in E$ can be added. It is called maximum cardinality matching $Mcm$ if it contains the maximum possible number of edges. If $\forall v \in V$, $\exists e \in Mm$, $v \in e$, then it is called perfect matching $Pm$. Hence, it is clear that $Pm \subset Mcm \subset Mm \subset m$. The perfect matching on $G(V, E)$ is actually a special case of factorization $G$, in which an $r$-factor is defined as a union of $r$-regular spanning subgraphs of $G$. Hence, perfect matching is also termed as 1-factor of $G$ [1]. It is also referred to as complete dimer cover in statistical mechanics [2], and Kekulé structure [3] in quantum chemistry. The necessary and sufficient condition for a graph to have a perfect matching is given by Tutte and Lovász [4–5]. The total number of perfect matchings on $G$ is denoted by $\psi(G)$. Every perfect matching will have a unique collection of disjoint edges of $G$. Our primary concern in this work is to develop an algorithm that counts the number of perfect matchings $\psi(G)$, as well as to generate the edges in each of them.

Matchings in general, and perfect matchings in particular have interesting applications [2,6–10]. In the chemical context, $\psi(G)$ is referred to as Kekulé structure count $K$, and it is widely used in the theory of conjugated hydrocarbons [8–10]. Enumeration of Kekulé structures received enormous attention over the years [11–22], and we will mention only the papers relating to different methods developed to count $K$. For comparison, algorithms reported in the mathematical literature for finding maximal matchings on graphs are mentioned in the next section.

One of the authors (M.M.B) wishes to express his gratitude to UGC, India for financial support.
[1]Author to whom correspondence should be addressed

There are several interesting algorithms available in the mathematical literature for finding maximal matchings of graphs (especially for bi-partite graphs) but all of them generate only a single maximal cardinality matching [23,24]. Recently, parallel algorithms for finding maximal cardinality matching have also been reported [25,26]. In the chemical literature, several methods on the enumeration of Kekulé structures have been reported. They can be classified as graph theoretic recursive methods [11], determinantal methods [12], reduced graph method [13], graph decomposition method [14,15], transfer matrix method [16], Herndon's method [12], methods based on the John-Sachs theorem [17–19], and computer assisted methods [20–22].

A cursory glance of the above methods clearly reveal that none of them applies to a general graph, i.e., each one applies to graphs of special kinds only [27,28]. Mostly, the methods exploit the inherent combinatorial regularity of graphs to which they apply, and hence, are highly efficient. Some of the existing methods provide closed formulae for Kekulé structure count, while some lead to hand-computable procedures. Only a few need computerization for enumeration. As this problem of enumeration and generation of all perfect matchings has not so far been addressed for a general graph, either in the mathematical literature or in the chemical literature, we do it here. We use a search technique that would be normally preferred, especially when no combinatorial regularity is assumed to design the algorithm. The problem explodes combinatorially as the graph size increases, and to partially offset this, we employ Artificial Intelligence techniques. Finally, we also show that, taking few examples, if suitable heuristics are used in the combinatorial exploration, the time complexity of our algorithm can be reduced.

## 2. ALGORITHM DESCRIPTION

### 2.1. Potential Graph Representation using Relations

Let $G(V, E)$ be a finite graph. To explore matching in a systematic manner, we define two binary relations on the labeled graph under consideration.

(1) The first relation, $\rho$, produces a set of ordered pairs (symbols) $\langle x, y\rangle$, such that

$$\{\langle x, y\rangle \mid \forall x, y \in V \ (x \text{ adj } y) \wedge (x < y)\}, \tag{2.1}$$

where $(x \,\text{adj}\, y)$ is true iff there is an edge between $x$ and $y$.

(2) The second relation $\zeta$ is defined on $\rho$ as

$$\{\langle x, y\rangle \mid \exists \langle x, y\rangle \in \rho, \ \exists i \in D_\rho, \ x = i\}, \tag{2.2}$$

where $D_\rho$ is the domain of $k$.

Since $\zeta$ is an equivalence relation, it splits the set of ordered pairs produced by $\rho$ into partitions. A matching $u$ is the string $u_1, \ldots, u_n$ of any finite number of symbols (edges) with length $n$ which are mutually disjoint. Given the matchings $u, v \in L$,

$$\begin{aligned} v &= \text{prefix}(u) && \text{iff} \quad \exists w \in L, && u = v\,w, \\ v &= \text{suffix}(u) && \text{iff} \quad \exists w \in L, && u = w\,v, \\ u &= \text{sub-matching}(v) && \text{iff} \quad \exists x, y \in L, && v = x\,u\,y. \end{aligned}$$

A partition $\zeta_i$ is said to be disjoint to a matching, if the vertex labeled $i$ is not involved in the matching. Since no two edges are the same in a given matching, it can be considered equally as a set.

## 2.2. Operator Specification

The set of all matchings $L$ over the finite alphabet $\zeta$ forms a language $L$, which can be defined inductively as follows.

(1) The null string $m$ and $\zeta$ are in $L$.

(2) If $u, v \in L$, the concatenation $u$ and $v$, is the matching $w$ of length $|n| + |m|$, obtained by the disjoint-set-union of $u$ and $v$, such that

$$w = \begin{cases} u\,v, & \text{iff } \forall\langle a,b\rangle \in u, \forall\langle x,y\rangle \in v, \{a,b\} \cap \{x,y\} = \emptyset, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

(3) A matching is in $L$ only if it is obtained by the application of clauses (1) and (2).

It can be easily seen that $L$ is closed under a disjoint-set-union operation for a given alphabet $\zeta$. Hence, it can be deduced that the total number of perfect matchings is equal to the cardinality of the set $PM$, which is given by

$$PM = \{u \in L \mid N \text{ div } 2 = |n|\}, \tag{2.3}$$

where $N$ is the total number of vertices in graph $G$. $PM$ is a proper subset of $L$ and its members are all perfect matchings. Now, we present a significant nature of this enumeration problem.

THEOREM 1.1. *The problem of generating all the perfect matchings of a general graph $G$ is in $NP$.*

PROOF. A nondeterministic algorithm for generating perfect matchings has simply to check that any collection of $N$ div 2 edges satisfies the condition of disjointness in an instance of the problem. It is a simple matter to construct a polynomial time NDTM (Non Deterministic Turing Machine) to do this. Thus, it is in $NP$. ∎

It is this $NP$ nature of the problem, which restricts the applicability of the earlier polynomial time algorithms to some specific domains [16–22], and hence, an effective generalized code for this problem deserves the attention of Artificial Intelligence.

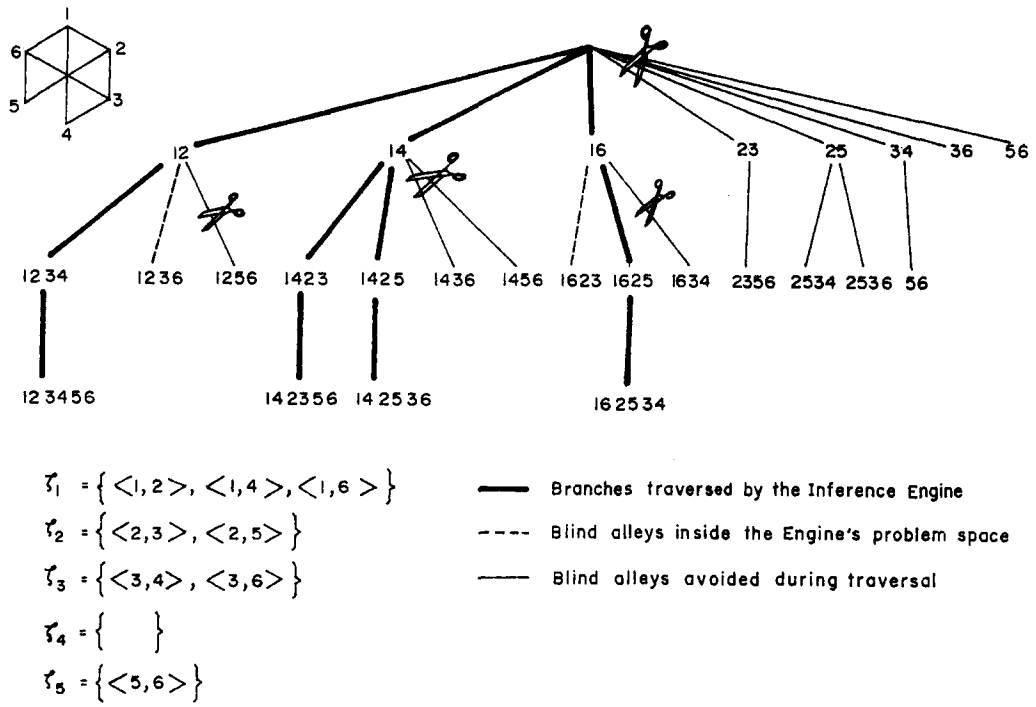
## 2.3. Structure of Inference Engine

The partitions of $\zeta$ are sorted and represented by a linked list. The next step is to derive an Inference Engine (`IE`) to generate the matchings of various lengths, from the symbols of $\zeta$. The inference control strategy coupled with the alphabet $\zeta$ as database, and perfect matchings as goals comprise a search mechanism for the Kekulé structure count. Thus, all the matchings of $L$ that are generated by the production system fall in a tree domain [29] (Figure 1). The nodes at a depth $d$ correspond to a matching $u$. A branch rooted at the node $u$ is a subtree whose domain is given by $\{v \in L \mid \text{prefix}(v) = u\}$. Since the width of the search tree is greater than the depth, the preferable search strategy is depth-first in order to minimize the memory requirements.

Given the initial instance with the matching being null, the algorithm invokes a copy of the current algorithm, but with a different instance to solve. This phenomenon, called recursion, is the very heart of the search mechanism on which the whole `IE` is designed quite compactly. The structure of the recursive function is like

```
function match (u ∈ L)  :=  if    ⟨disjoint test⟩
                            then  ⟨recursive rule⟩
                            else  ⟨perfect test⟩
```

The ⟨`disjoint test`⟩ is a predicate that tests whether the matching $u$ and some edge are disjoint. If so, the ⟨`recursive rule`⟩ expression generates the matching that is descending from




$\zeta_1 = \{ \langle 1,2 \rangle, \langle 1,4 \rangle, \langle 1,6 \rangle \}$

$\zeta_2 = \{ \langle 2,3 \rangle, \langle 2,5 \rangle \}$

$\zeta_3 = \{ \langle 3,4 \rangle, \langle 3,6 \rangle \}$

$\zeta_4 = \{ \}$

$\zeta_5 = \{ \langle 5,6 \rangle \}$

Figure 1. The search tree of the graph that is traversed during enumeration. Every node in the tree represents a specific matching. The scissors drawn on the path denote the points of heuristic scissoring.

the currently assigned matching, assigns newer values to the formal arguments, and calls itself recursively. If not, the ⟨**perfect test**⟩ expression checks whether the matching is perfect and provides the results. The implementation of this technique involves the concept of stack. Each time the function refers to itself, the program executing it saves on this stack a complete set of information regarding the matching it is assigned within the formal argument, the position of the pointer in the partition list, and some other relevant information. Such a collection of information is referred to as a *frame*. The **IE** is invoked with null matching, beginning its search from the first member, i.e., $\zeta_1$, of the partition list. Then, it generates the new matchings and computes other argument values by searching for a disjoint edge in the rest of the list. If an edge satisfies the ⟨**disjoint test**⟩, control transfers to the recursive rule code, where a new frame is built and a new call to the function takes place. Failure of the ⟨**disjoint test**⟩ in all partitions leads to the ⟨**perfect test**⟩ rule code which infers the desired answer. Further, it checks the status of this internal stack. If the stack is empty, the answer is returned directly. If not, the top frame is popped back, and the code is restarted at the point where it is suspended itself; having the just-computed results, it searches for other edges that satisfy the ⟨**disjoint test**⟩. Again, when this code completes itself, it tests the stack before quitting. A nonempty stack triggers another popping sequence.

The tree was searched systematically 'depth-first' and then from left to right descending the level as far as possible, generating the matchings as deep as possible before meeting an *impasse* and back-tracking. The leaves of depth $N$ div 2 correspond to perfect matchings.

## 2.4. Search Computational Complexity

The **IE** had to tranverse the entire tree searching for the leaves corresponding to perfect matchings. This problem cannot be easily described in a form that leads to immediate mathematical derivations for the number of perfect matchings. Computation of problem space $L$ may be attempted by exhaustive search as described above, in which case the **IE** demands $O(n!)$ time for a graph with $n$ vertices. Since this is quite exhaustive and time consuming, it is seldom possible

except for smaller graphs. Hence, an ideal IE should explore only a subset $L'$, of the problem space $L$, which is given by

$$L' = \{u \in L \mid \exists v \in PM,\ u = \text{prefix}(v)\}. \tag{2.4}$$

This requires scissoring certain branches in the search tree whose root is $u \notin L'$. Identification of these branches are not straightforward since the IE is not provided with *a priori* knowledge about $L'$. But a several of these branches can be scissored by an intelligent inquiry about the nodes when they are generated. This is tactically solved by heuristics, which is deduced using certain properties of $\zeta$ described in the next section.

### 2.5. Representation Characteristics

The structure we have chosen for graph representation has certain important characteristics which can be used to improve the efficiency of search. They help in identifying and avoiding blind alleys in the pathway of the inference engine.

THEOREM 2.1. *For any $m \in L$, the partitions of edges in $m$ are all distinct.*

PROOF. Since $\forall i \in n$ $(\forall u \in \zeta_i,\ i \in u)$, no two edges of a partition are disjoint, and hence, form a matching. ∎

THEOREM 2.2. *For any depth $D$, the matchings arising from the first found disjoint partition (FFDP) form a superset of $L'$.*

PROOF. For a graph $G$ with $N$ vertices ($N$ is even), the perfect matching of $G$ contains $\frac{N}{2}$ number of edges, which requires all the vertices of $G$. Let $\zeta_j$ be the FFDP of the list at a depth $D$. Since, from (2.1) it can be deduced that an edge $\langle x, y\rangle$ will not be found in partitions $\{\zeta_i \mid (i > x)\}$, the matchings arising from these partitions will not contain the vertex $J$. Hence, they cannot include any perfect matching. ∎

COROLLARY 2.3. *If the first found disjoint partition of matching $u$ contains no disjoint edge, then $u \notin L'$.*

PROOF. By Theorem 2.2, it is proved that the vertex $j$ of the first found disjoint partition $j$ cannot be found from the remaining partitions of the list. Hence, absence of any disjoint edge in the first found partition implies that the current path is a blind alley. ∎

### 2.6. Scissoring of Blind Alleys

The above facts can be used successfully to design powerful heuristics. Partitioning of $\rho$ by $\zeta$ helps to reduce the number of edges to be tested for disjointedness. Using Theorem 2.1, the IE can safely avoid searching in the same partition. Using Theorem 2.2, it can refrain from searching for a disjoint edge in all, other than the FFDP. Hence, the IE will backtrack rapidly to the previous frame terminating its search in the current frame. In cases where the FFDP is empty or does not possess any disjoint edge, the IE infers that the current path is a blind alley and reverts back without continuing further. Hence, the number of partitions that are to be explored is reduced to unity. This criterion actually helps to avoid many of the blind alleys, thereby drastically decreasing the branching factor. This is clearly illustrated with the example in Figure 1. Earlier branching factor is a function of $N$ and now it is the cardinality of the partition which is a constant. In other words, *a priori* time complexity is reduced from $O(n!)$ to $O(n^k)$, where $k$ is a constant.

As we are addressing the problem in a more general context, we feel it convenient to use search to develop the algorithm. The algorithm discussed above is given in Table 1. Due to the recursive nature of the algorithm, it can be implemented in any high level language that supports recursion with relative ease.

Table 1.

Algorithm in *pseudo* for the generation of all perfect matchings on a general graph.

```
PROCEDURE PERFECT MATCH(      POSITION : pointer to current partition
                          VAR MATCH    : matching)
  REPEAT
    FOR ALL U ∈ MATCH DO
      BEGIN IF POSITION ∉ U THEN
        BEGIN FFDP = ζPOSITION
          FOR ALL ⟨X,Y⟩ ∈ FFDP DO
            BEGIN FOR ALL U ∈ MATCH DO
              BEGIN IF Y ∈ U THEN
                BEGIN Add edge U to MATCH
                  IF NOT MATCH is perfect matching THEN
                        PERFECT MATCH (POSITION, MATCH)
                        ELSE output MATCH
                  Remove U from MATCH
                ENDIF
              ENDFOR
            ENDFOR
    POSITION ← POSITION + 1
  UNTIL FFDP is found or the list is exhausted.
END.
```

# 3. DOMAIN SPECIFIC ENHANCEMENTS

Though the above discussed heuristics closes many blind alleys, some of the matchings that are not in $L'$ are left unidentified before traversing. For an arbitrary graph of considerable complexity, such search based enumeration is bound to be combinatorially exploding, and to partially offset this, several Artificial Intelligence techniques can be used. It should be noted that for symmetric and regular graphs the complexity of the algorithm can be considerably reduced by developing special purpose heuristic functions and incorporating domain specific knowledge in the search skeleton. Various ways of improving algorithmic time complexity for suitable problem domains are given below.

## 3.1. Rote Learning

In highly clustered graphs and for graphs containing many even membered rings, many matchings have the same set of vertices though they belong to different set of edges. The total number of perfect matchings, $\{Pm \mid m = \text{prefix}(Pm)\}$ is the same for all these matchings. Hence, a learning procedure to find this similarity can drastically reduce the computing time. But this requires preserving those matchings and results during the entire search, which demands additional memory space.

## 3.2. Domain Specific Knowledge Bases

Another way of improving the time complexity is by the identification of isomorphic branches in the search tree. This is achieved when the information regarding edges leading to isomorphic branches are provided to the engine, either directly or through a program that identifies such branches. This can help the inference engine to refrain from traversing similar branches. Since this needs a very generalized code, it will be a lot easier to develop the code in any of the Artificial Intelligence dedicated languages like PROLOG. In the case of complete and complete bi-partite graphs, since edges of the same partition are all equivalent, the algorithm takes only polynomial time to arrive at the results.

### 3.3. Constraint Seeking

The number of blind alleys in the path can be reduced by seeking the existence of additional constraints if any, in the selection of branches. For sparse graphs, i.e., chemical graphs, the number of nonzero entries in their adjacency matrix is very small. Knowledge captures this kind of generalization to enhance the speed of the search. In this case, the sequential search through the partition list is overlooked at every frame in order to include certain edges of special category. This category of edges contains at least one of their vertex unique, which will not be found in the rest of the partition list. The vertices of these edges, hence, are compulsory members of any perfect matching that can be obtained from the matching of the current node. This technique improves the selectivity of search and is very suitable for graphs with few cycles.

## REFERENCES


1. J.A. Mchugh, *Algorithmic Graph Theory*, Prentice Hall, New Jersey, (1990).
2. E. Montroll, Lattice statistics, In *Applied Combinatorial Mathematics*, (Edited by E.F. Bechenbach), pp. 96–143, Wiley, New York, (1964).
3. N. Trinajstić, *Chemical Graph Theory*, Vol. II, CRC Press, Boca Raton, Florida, (1983).
4. W.T. Tutte, The factorization of linear graphs, *J. London Math. Soc.* **22**, 107–111 (1947).
5. L. Lovász, Three short proofs in graph theory, *J. Combinatorial Theory* **B19**, 111—113 (1975).
6. P.W. Casteleyn, The statistics of dimer on a lattice, *Physica* **27**, 1209–1225 (1961).
7. M.E. Fisher, Statistical mechanics of dimers on a plane lattice, *Phys. Rev.* **124**, 1664–1672 (1961).
8. S.J. Cyvin and I. Gutman, *Kekulé Structures in Benzenoid Hydrocarbons*, Springer-Verlag, Berlin, (1988).
9. S.J. Cyvin, J. Brunvoll and B.N. Cyvin, Topological aspects of benzenoids and coronoids, including "snowflakes" and "laceflowers", *Computers Math. Applic.* **17** (1–3), 355–374 (1989).
10. S.J. Cyvin and I. Gutman, Kekulé structures and their symmetry properties, *Computers Math. Applic.* **12-B** (3/4), 859–876 (1986).
11. M. Gordon and W.H.T. Davison, Theory of resonance topology of fully aromatic hydrocarbons, *J. Chem. Phys.* **20**, 428–435 (1952).
12. W.C. Herndon, Enumeration of resonance structures, *Tetrahedron* **29**, 3–12 (1973).
13. P. Křivka and N. Trinajstić, In *Mathematics and Computational Concepts in Chemistry*, (Edited by N. Trinajstić), Horwood, Chichester, (1986).
14. M. Randić, Enumeration of Kekulé structures in conjugated hydrocarbons, *J. Chem. Soc. Faraday Trans. 2* **72**, 232–243 (1976).
15. I. Gutman and N. Trinajstić, Graph theory and molecular orbitals, IV. Further applications of Sach's formula, *Croat. Chem. Acta.* **45**, 423–429 (1973).
16. E.H. Lieb, Solution of the dimer problem by the transfer matrix method, *J. Math. Phys.* **8**, 2339–2347 (1967).
17. W. He and W. He, On Kekulé structure and $P - V$ path method, In *Graph Theory and Topology in Chemistry* (Edited by R.B. King and D.H. Rouvray), pp. 476–483, Elsevier, Amsterdam, (1987).
18. I. Gutman and J. Cyvin, A new method for the enumeration of Kekulé structure counts of benzenoid and caronoid hydrocarbons, *Chem. Phys. Lett.* **136**, 137–140 (1987).
19. D.J. Klein and N. Trinajstić, Pascal recurrence algorithm for Kekulé structure counts of benzenoid and caronoid hydrocarbons, *J. Mol. Struct. (Theochem.)* **206**, 135–142 (1990).
20. Džonova-Jerman-Blažič and N. Trinajstić, Computer aided enumeration and generation of the Kekulé structures in conjugated hydrocarbons, *Comput. Chem.* **6**, 121–132 (1982).
21. J.V. Knop, K. Symanzki, N. Trinajstić and P. Křivka, Computer generation of all 1-factors for a class of graphs with all vertices of degree two or three, *Computers Math. Applic.* **10** (4/5), 369–382 (1984).
22. R.L. Brown, Counting of resonance structures for large benzenoid polynuclear hydrocarbons, *J. Comp. Chem.* **4**, 556-562 (1983).
23. J. Edmonds, Paths, trees and flowers, *Canadian J. Math.* **17**, 449–467 (1965).
24. H.N. Gabow, An efficient implementation of Edmonds algorithm for maximal matchings of graphs, *Journal ACM* **23**, 221–234 (1976).
25. K. Mulmuley, U.V. Vazirani and V.V. Vazirani, Matching is as easy as matrix inversion, In *Proc. $19^{th}$ Ann. ACM Sympos. on Theory of Computing*, pp. 345–354, ACM, Baltimore, MD, (1987).
26. A.V. Goldberg, S.A. Plotkin and P.M. Vaidya, Sublinear-time parallel algoritms for matching and related problems, *J. Algorithms* **14**, 180–213 (1993).
27. N. Trinajstić, D.J. Klein and M. Randić, On some solved and unsolved problems of Chemical graph theory, *Int. J. Quantum Chem. Quantum Chem. Symp.* **20**, 699–742 (1986).
28. H. Hosoya, Matching and symmetry of graphs, *Computer Math. Applic* **12-B** (1/2), 271–282 (1986).
29. M.M. Balakrishnarajan and P. Venuvanalingam, General method for the computation of matching polynomials of graphs, *J. Chem. Inf. Comput. Sci.* (to appear).